SWAP: Parallelization through Algorithm Substitution

By explicitly indicating which algorithms they use and encapsulating these algorithms within software components, programmers make it possible for an algorithm-aware compiler to replace their original algorithm implementations with compatible parallel implementations, or with the parallel implementations of compatible algorithms, using the so-called specification compatibility graph (SCG). Along with the SCG, a software environment is introduced for performing algorithm-aware compilation.

Hengjie Li
Wenting He
Yang Chen
Institute of Computing Technology, CAS

Lieven Eeckhout
Ghent University

Olivier Temam
INRIA Saclay

Chengyong Wu
Institute of Computing Technology, CAS

Automatic compiler-based parallelization isn't yet mature enough to tackle a broad range of programs, despite significant recent advances,[1,2] so parallelization is still likely to involve the programmer. As a result, the current key research issue is to make it as easy as possible for the programmer to parallelize applications. The most favored approach so far is to propose parallel environments that hide as much of the parallelization complexity as possible from the user. For instance, Intel's Threading Building Blocks (TBB)[3] and Cilk[4] facilitate parallelization by reducing it to the intuitive notion of splitting a task into subtasks, shielding the user from mapping, scheduling, and other issues, all of which are handled by the runtime environment. However, the user is still involved in the complex tasks of finding parallelism and managing dependences.

Because the potential gains of parallelization are high, it's reasonable to expect the user to devote some effort to transforming the program. However, most programmers aren't proficient in parallelization techniques or processor architectures, so the necessary transformations should remain in the realm of their expertise and should require as little architecture knowledge as possible.

One existing approach matches these characteristics: parallel libraries. To use parallel libraries, the only effort required from the user is to modify the program so that it matches the library's I/O interface. At the cost of that transformation effort, any nonexpert user can exploit a parallel library's performance benefits. However, there are few parallel libraries—for example, Parallel Standard Template Library (STL), Standard Adaptive Parallel Library (Stapl),[5] Multicore Standard Template Library (MCSTL),[6] and Thrust (http://code.google.com/p/thrust). There is a vast set of tasks that programmers want to perform, and few of the algorithms implemented in a given program are likely to be covered by existing parallel libraries. For information on other approaches, see the "Related Work on Components and Parallelization" sidebar.

We start from a simple postulate: although few programmers are proficient in

Published by the IEEE Computer Society

Related Work on Components and Parallelization

The notion of software components is broadly used in software engineering for allowing developers to work independently on large projects[1] and seamlessly modify and replace program parts. Component frameworks, such as JavaBeans,[2] .Net,[3] or the Fractal Open Component Model[4] usually define and enforce programming contracts (strict encapsulation of both code and data, explicit input and output interfaces, and so on). Here, we mainly retain the notions of encapsulation and explicit interfaces.

To a certain extent, some parallel environments implicitly or explicitly leverage the notion of components. For instance, both Threading Building Blocks[5] and Cilk[6] implicitly encapsulate tasks to be parallelized. Charm++ explicitly and strictly encapsulates tasks in component-like structures because they're meant to be executed on a distributed-memory machine and to communicate via message passing.[7] Linderman et al. also propose to view programs as a set of individual components and efficiently map them on multiple cores.[8] However, the parallelization is again done by the programmer using a MapReduce-like approach.

Thomas et al. propose to encapsulate tasks such as sorting and matrix-multiply into the software containers provided by STAPL (the Standard Template Adaptive Parallel Library)[9] to explore the best parallel versions of these algorithms.[10] The STAPL framework embodies a number of compatible algorithms and features a prediction model constructed through machine learning to select algorithms. Pan et al. want to facilitate the use of the different existing parallel libraries to help develop parallel applications.[11] So, they propose low-level primitives and common interfaces for developers to use multiple parallel libraries within their applications.

Like Petabricks,[12] we promote an algorithmic-centric approach to programming. However, in Petabricks, these versions are written and embedded within the program itself by the programmer. One of our fundamental hypotheses is that most programmers are either unwilling or unable to write efficient parallel programs. Because in Petabricks alternative implementations are not sought externally, there is also no attempt at characterizing the compatibility of algorithms and implementations.

References

1. K. Lau and Z. Wang., ''Software Component Models,'' *IEEE Trans. Software Eng.,* Oct. 2007, pp. 709-724.
2. L. DeMichiel,, *Enterprise JavaBeans Specification, Version 2.1,* Sun Microsystems, Nov. 2003.
3. A. Rasche and A. Polze, ''Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET,'' *Proc. 6th IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing* (ISORC 03), IEEE CS, 2003, p. 164.
4. E. Bruneton et al., ''The FRACTAL Component Model and Its Support in Java,'' *Software: Practice and Experience,* Sept. 2006, pp. 1257-1284.
5. J. Reinders, *Intel Threading Building Blocks,* O'Reilly & Associates, 2007.
6. R. Blumofe et al., ''Cilk: An Efficient Multithreaded Runtime System,'' *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPOPP 95), ACM, 1995, pp. 207-216.
7. L. Kale and S. Krishnan, ''CHARM++: A Portable Concurrent Object Oriented System Based on C++,'' *Proc. 8th Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 93), ACM, 1993, pp. 91-108.
8. M. Linderman et al., ''Merge: A Programming Model for Heterogeneous Multi-core Systems,'' *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2008, pp. 287-296.
9. P. An et al., ''STAPL: An Adaptive, Generic Parallel C++ Library,'' *Proc. 14th Int'l Conf. Languages and Compilers for Parallel Computing,* Springer, 2001, pp. 193-208.
10. N. Thomas et al., ''A Framework for Adaptive Algorithm Selection in STAPL,'' *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 05), ACM, 2005, pp. 277-288.
11. H. Pan, B. Hindman, and K. Asanovic, ''Composing Parallel Software Efficiently with Lithe,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 10), ACM, 2010, pp. 376-387.
12. J. Ansel et al., ''PetaBricks: A Language and Compiler for Algorithmic Choice,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 09), ACM, 2009, pp. 38-49.

architectures and parallelization techniques, there are probably a few expert programmers capable of writing, or who even have already written, parallel versions of some of the key algorithms. Then, the key difficulty is to leverage that expertise and let nonexpert programmers benefit from existing parallel versions of algorithms. The key concept behind the proposed SWAP software environment is that nonexpert programmers specify the semantic requirements of the algorithms in the program, and the SWAP framework then automatically swaps algorithms and implementations to optimize performance on a given processor architecture. By doing so, SWAP improves both performance and productivity.

To achieve this goal, we build on three notions previously developed in the literature: viewing programs as a composition of

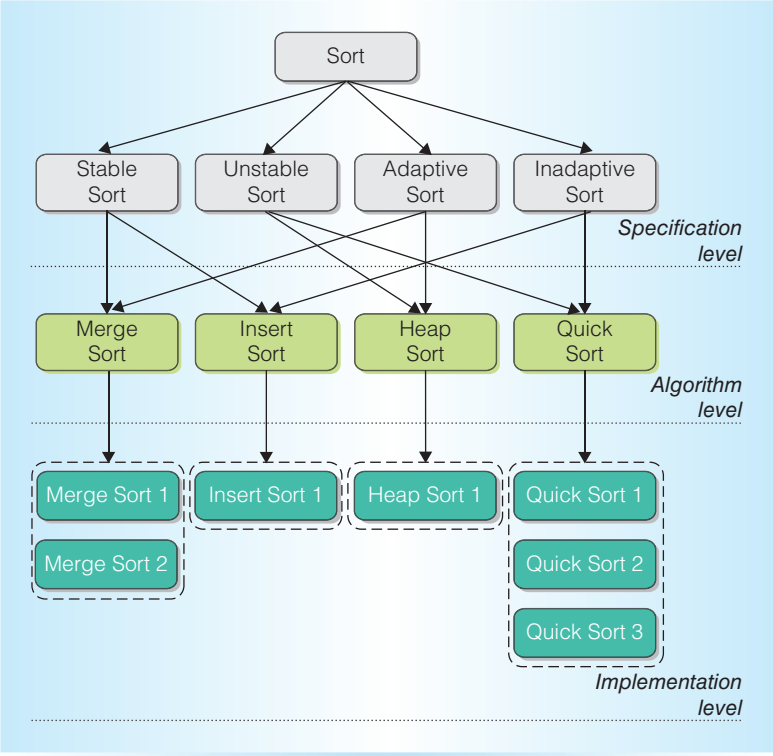

Figure 1. Specification compatibility graph (SCG). There are three levels of abstraction: the specification, the algorithm, and the implementation.

algorithms with glue code around them;[7] *community-based development*—that programs are increasingly developed by bringing together code parts from many different sources (such as libraries, STL, PHP functions, or even code snippets found using Google Code Search);[8] and the notion of *software components*, tightly encapsulated code parts with explicit communication interfaces, which are popular in software engineering because they allow several programmers to independently develop different code parts of a large program.[9,10]

Our approach

We aim to make the compilation process algorithm-aware by writing (or rewriting) programs as compositions of algorithms wrapped into software components containing explicit algorithm information. The compiler parses the algorithm information and looks for compatible implementations in a software repository; then the compiler instantiates the selected algorithm implementations and generates the final code. This repository is called by, but is independent from, the compiler. It can be independently populated by expert programmers; thus, it constantly evolves, and the newly added algorithm implementations are immediately available to programmers without any compiler or code modifications.

Our approach's key technical challenge is to determine whether two algorithm implementations can be substituted. This means that the two algorithms must realize the same task, and their implementations (that is, their interface and platform requirements) must be compatible. Therefore, we propose the specification compatibility graph (SCG), which captures all compatibility information within a single structure, such that it is comprehensible for the compiler. The SCG is the key enabler for the proposed SWAP framework, which lets programmers automatically substitute algorithm implementations. In contrast, traditional software libraries require the programmer to perform the exploration. The SWAP framework currently supports C and C++ programs: the programmer breaks up a program in different components and provides compatibility information using pragmas; the compiler doesn't require modification, because a precompiler processes the framework's syntax. The framework then automatically searches for and substitutes alternative algorithms and implementations.

Specification compatibility graph

The SCG captures the compatibility of algorithms and algorithm implementations.

Compatibility of algorithms

We first explain how the SCG captures the compatibility of algorithms and how it depends on both qualitative and quantitative characteristics.

Qualitative algorithm characteristics. We distinguish three levels of abstraction for characterizing a program (see Figure 1): the *specification* (the definition of what the program must be doing, such as sorting a list of data elements), the *algorithm* (the method used to perform that task, such as Quick Sort or Merge Sort), and the *implementation* (programming that method, such as multiple different implementations of Quick Sort).

The specification is the only true constraint imposed by the programmer. In theory, any program abiding by that specification is compatible, independent of the algorithm or implementation used. For instance, once we know that the target task is sorting, we can potentially use either Quick Sort or Merge Sort. Thus, knowing the task specification opens up many alternative algorithms and implementation choices.

In practice, though, just knowing the specification isn't enough. Often, a programmer needs or wants to use a specific category of algorithms because they share certain properties for the task at hand. For instance, certain sorting algorithms are deemed *stable* because they maintain the relative order of records with equal values. Therefore, the information should not just be *sorting*, but sorting and stable or unstable. At the same time, a user not concerned about the stability of a sorting algorithm can just specify sorting and use whatever algorithm is available. However, there are many such attributes for any task. For instance, sorting algorithms can also be *adaptive* or *inadaptive* (a sorting algorithm that takes advantage of the input order to speed up sorting). We can't expect to embed all such task-specific attributes for all possible tasks within a compiler; there are too many attributes, and they evolve continually.

However, a compiler doesn't need to know task-specific information, such as stable versus unstable sorting. It only needs to know whether an algorithm is compatible with a set of programmer-specified constraints. Algorithms, attributes, and implementations can be organized into a hierarchy, which can be represented as the graph of Figure 1. In the graph, the nodes correspond to any type of abstraction information (task specification, algorithm, attribute, implementation), and the edges characterize the abstraction relationship (''more abstract than''). We call such a directed acyclic graph an SCG. This graph can deliver the compatibility information to a compiler in a task-independent manner. Using this graph, an algorithm is deemed compatible with a set of programmer-specified constraints simply if it belongs to the intersection of the spanning trees of all these constraints, whatever they correspond to. For instance, Merge Sort and Insert Sort are in the spanning tree of Stable Sort; but only Merge Sort belongs to the intersection of the spanning trees of Stable Sort and Adaptive Sort.

```
...
cpt_StableSort(...);
...
//cpt_pragma: hash strength >= 52
cpt_CryptographicHash (...);
...
```

Figure 2. Component reference and annotation for specifying a quantitative characteristic. The `cpt_` prefix distinguishes component code from user code; the `cpt_pragma` denotes quantitative characteristics for the component.

How does a programmer use the SCG? In the code, the programmer specifies the graph node with the highest abstraction level compatible with the target task, as shown in Figure 2, and references the corresponding prototype in the program (we use a special prefix `cpt_` to distinguish component code from user code). Then, the compiler knows that it can use any instance located in the specified node's spanning tree. Although nodes are labeled with explicit abstraction information (such as stable or unstable sort), this human-readable information is provided only for the programmer. The compiler uses only the relative relationship between the different nodes, not what each node stands for. The SWAP framework then automatically explores alternative compatible algorithm implementations and searches for the best one.

Quantitative algorithm characteristics. We must distinguish between qualitative and quantitative algorithm characteristics. Qualitative characteristics, such as sorting stability and adaptivity, are expressed in the compatibility graph. Quantitative characteristics can't be expressed in the same way, owing to the possibly infinitely large number of options. For instance, hashing algorithms, used for cryptography or compression, are characterized by their strength. A strength

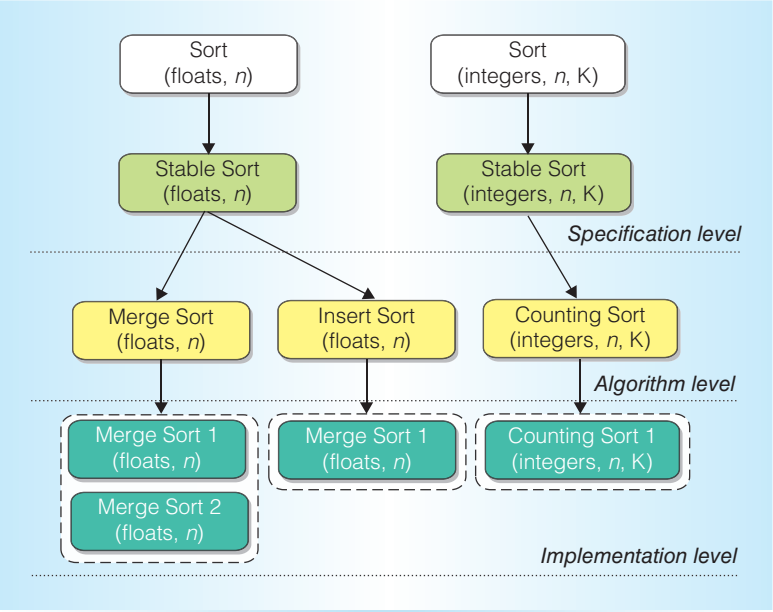

Figure 3. Compatibility of interfaces. The SCG expresses compatibility between interfaces through edges between nodes.

of 2^m means that a collision can occur between two hashed values after 2^m operations. (Operations are defined as bitwise operations such as add, and, or, XOR, and rotate; the number of such operations is used as the measure of a hashing algorithm's strength.) There are many well-known hashing algorithms—for instance, $m = 25$ for MD5, $m = 39$ for SHA-0, and $m = 52$ for SHA-1. So, we could create SCG nodes corresponding to different intervals, such as [20; 40] and [40; 60], and organize the algorithms accordingly. However, these intervals would be arbitrary, because there's no clear way to break down the algorithms according to such quantitative characteristics.

Nevertheless, the programmer should have a way to restrict the algorithm selection on the basis of these quantitative characteristics. For that purpose, when the programmer selects the abstraction level, we also provide the ability to specify quantitative characteristics using annotations, as Figure 2 shows (see `cpt_pragma`). These annotations are component specific, and they're part of the component documentation available in the repository. Within the database, the existing component implementations optionally specify these quantitative metrics. If a programmer requests certain algorithm characteristics, but the database contains no information about these characteristics for some of the algorithm implementations, they are deemed inappropriate for the target task, and are therefore ignored.

Compatibility of implementations

Being able to express compatibility of algorithms isn't sufficient; we also need the ability to express the compatibility of the algorithm implementations. The two essential aspects are the implementation interfaces and platform dependence.

Interfaces. Besides performing the same task (compatibility of algorithms), the algorithm implementation should use the same inputs and produce the same outputs (compatibility of implementations). This requires not just type checking of the interface's inputs and outputs, but checking that the *semantics* (the role of the inputs and outputs in the algorithms) are the same; again, such semantic checking is often beyond a compiler's reach.

There's no easy way for the programmer to express this information in a manner the compiler can comprehend. This information is usually found in variable names and human-readable comments only. More sophisticated program specification techniques, such as Unified Modeling Language (UML),[11] are meant to formally express such information, but programmers seldom use them.

However, it's possible to embed this information in the SCG so as to detect the compatibility of two implementations. As we mentioned earlier, the SCG expresses implicit and relative abstraction relationships, as opposed to explicit and absolute abstraction information. We apply the same principle to the implementation interfaces. We never explicitly define the nature, type, or data structure of the interface's input and output parameters. We simply organize the implementations, within the graph, according to their compatibility. Consider the example in Figure 3. At any abstraction level, beyond the qualitative and quantitative algorithm characteristics, we're now also distinguishing nodes by their I/O interface. This includes each parameter's data type information as

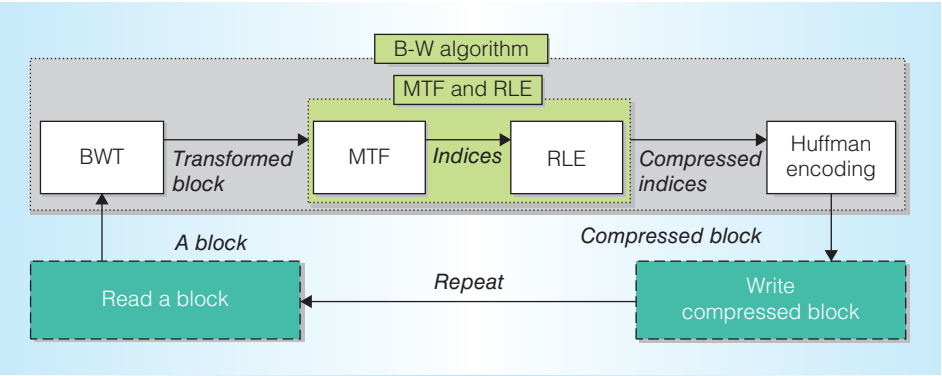

Figure 4. Block diagram of BZIP2; componentized algorithms are shown in white. BZIP2 implements the B-W algorithm, which consists of four components: Burrows-Wheeler Transform (BWT), move to front (MTF), run length encoding (RLE), and Huffman encoding.

well as its semantics. For instance, Merge Sort, Insert Sort, and Counting Sort are all stable sorts. Consider the case in which the library contains an implementation of Merge Sort and one of Insert Sort, which both take as inputs a sequence of floating-point data and the number of data elements, and an implementation of Counting Sort, which takes as inputs a sequence of unsigned integers and the maximum integer value. Then, there are two possible interfaces for stable sorts, each with a set of parameters with certain data type and semantics. As a result, we create two Stable Sort nodes, as Figure 3 shows.

Platforms. An algorithm implementation can be used on the target platform if the hardware (for example, single-instruction, multiple-data [SIMD] support) and software resources (for example, the TBB framework) it requires are available. As a result, much like compiler code generators require an architecture description file, our precompiler requires a platform description file that lists the available hardware and software resources; the database also records the hardware and software resources that each component implementation requires.

Program decomposition into components

A key principle of our approach is that many programs can be decomposed into a set of components corresponding to various well-identified algorithms; we call this process *componentization*.

Componentization

The notion of software components is broadly used in software engineering.[12] However, our implementation of components is light and only remotely related to formal component frameworks. From software components, we essentially retain two key notions: tight encapsulation and communications interfaces.

Tight encapsulation means that the algorithm code contained in the component has no program side effect: it can reference only internal data or data provided in the communication interface. Furthermore, there is no naming conflict with the rest of the program (using either a special name suffix in C, or a dedicated namespace in C++; we use the former for now). We can't use global variables to communicate between components and the rest of the program, unless they're (redundantly) passed as parameters. The component takes the form of one or several C functions or C++ classes, except that the encapsulation contract is tighter than usual.

We illustrate program componentization (wrapping program parts within components) with the SPEC CPU2006 version of BZIP2 (see Figure 4), in which each block corresponds to a component, and white blocks correspond to components substituted by SWAP. The main modifications correspond to algorithm encapsulation and modification of the algorithm interfaces. For instance, we modified 25 lines to encapsulate move to front (MTF) and run length encoding (RLE), 42 lines to adapt the interface of

Table 1. Benchmarks considered in this article, along with the number of components, number of source code lines, the number of modified source code lines, and the fraction of source code modified during componentization.

Benchmark	Domain	Components	Source code lines	Modified source code lines	Source code modified during componentization (%)	Datasets
ASTAR (SPEC 2006)	Computer games	1	5,842	2	0.03	Ref
BZIP2 (SPEC 2006)	Data compression	4	8,293	175	2.11	Ref.
CJPEG (MiBench)	Image compression	1	26,950	34	0.13	Gray 16-bit linear image
DEDUP (PARSEC)	Enterprise storage	5	3,683	121	3.29	Simlarge
LIBQUANTUM (SPEC 2006)	Quantum computer	6	4,357	157	3.60	Ref
VPR (vpr 5.0)	Field-programmable gate array (FPGA) placement and routing	1	40,197	141	0.35	Stdev000
BSDIFF	Linux utility	2	608	34	5.59	Two versions of libruby-static.a
FREQMINE (PARSEC)	Data mining	1	2,710	41	1.51	Simmedium
SCOTCH	Graph partitioning	3	12,040	54	0.45	Ncvxqp5

Burrows-Wheeler Transform (BWT), and 108 lines to adapt the interface of Huffman Encoding. Table 1 lists the other benchmarks considered in this study. The total number of source code lines that had to be modified varied between two and 175, and was less than 5.6 percent of the total program.

Parallelization

As we explained earlier, sequential algorithm implementations can be automatically replaced by parallel implementations. Typically, parallelizing applications requires careful consideration of the interplay (locks, synchronizations, and so on) between the different program parts. However, this isn't the case in the proposed approach, for two reasons: first, by definition, the components are encapsulated (that is, they don't modify global data structures), and, second, they maintain the sequential flow of the components composing the program (that is, components don't execute in parallel). As a result, the parallelization occurs within the component, where, on the other hand, expert programmers have implemented parallelism with the usual care for interplay between threads.

SWAP: An algorithm substitution framework

Figure 5 shows the overall SWAP framework.

Precompiler

Our current implementation targets C and C++ programs and components. Our method for referencing components and specifying annotations requires no special syntax, so the final program can be compiled using standard C or C++ compilers; we developed our precompiler as a front end to GCC.

As Figure 5 shows, the precompiler has four tasks: parsing the program to identify component references and annotations; reading the SCG stored in the database to check for matching specification, algorithm, or

implementation nodes; selecting the appropriate implementations (code and prototype) from the repository; and triggering the final compilation.

Database

The database consists of a set of tables that describe the SCG, essentially a Nodes and an Edges table. Table Nodes also indicates whether a node corresponds to an implementation in the software repository or to a more abstract node such as Stable Sort, for instance.

Repository

The repository is external to the precompiler so that it can be progressively augmented with new implementations or new algorithms as they become available, letting the programmer immediately benefit from them. Note that the precompiler can easily accommodate multiple repositories rather than just one.

In theory, for each newly uploaded algorithm implementation, the SCG maintainers must manually inspect the algorithm's attributes (such as stable or unstable, adaptive or inadaptive for sorting, and so on), and the interface parameters' semantics (such as the set of data to sort, vectors, or lists), and decide where this algorithm implementation would fit within the SCG. To partially automate this process, we request limited information from the contributing expert programmer: to select among a progressively expanding list of algorithm characteristics and interface semantics.

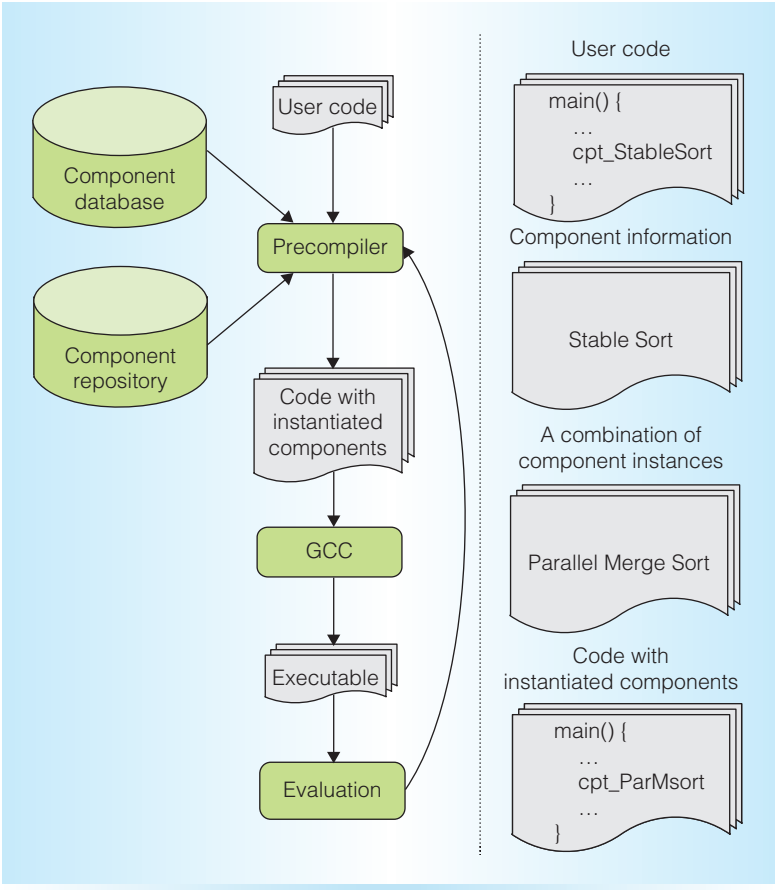

Figure 5. SWAP framework and overall process. The flow using the SWAP framework is shown on the left; the right side illustrates how user code is transformed from abstract components into instantiated components.

Experimental results

After describing our experimental methodology, we evaluate the overhead of componentizing programs, the performance benefits of substituting algorithms, and the impact of algorithm substitutions on algorithm-specific metrics.

Methodology

Table 1 lists the benchmarks and their inputs. No particular criterion guided our selection, except for our relative familiarity with some of the algorithms used in the benchmarks. For DEDUP (a benchmark taken from the PARSEC benchmark suite), we use the sequential version for the baseline performance, which is faster than the single-thread parallel version. Table 2 summarizes the componentization for all the benchmarks considered in this study. We conducted our experiments on an Intel Xeon E5430 quad-core processor, with a 12-Mbyte Level-2 (L2) cache and 16 Gbytes of RAM. The compiler is GCC 4.4.2, and the operating system is a version of RedHat Linux (4.1.2-44). For all benchmarks, the performance is averaged over three runs. We exhaustively explore all combinations of algorithm implementations and report the best performance. All the algorithm implementations come from the public domain (they were extracted either from publicly available applications or libraries); see the "Sources of Alternative Component Implementations" sidebar for a listing of all alternative component implementations in Table 2. We voluntarily did not reimplement any of

Table 2. Components of each benchmark and variants for each component.

Benchmark	Components	Algorithms and implementations		No. of combinations
		Original	Alternative	
ASTAR	Shortest path finding	Astar	Parallel bidirectional Astar	2
BZIP2	BWT	Seward's BWT	Div-BWT	12
	MTF	Move to front	MTF-1, MTF-2	
	RLE	Run length encoding	—	
	Encoding entropy	Huffman	Arithmetic	
CJPEG	Discrete Cosine Transform (DCT)	Seq-DCT	SIMD-DCT	2
DEDUP	Chunking	Rabin-Karp fingerprints	Fixed-size chunking	48
	Chunk redundant identification	Identifier	—	
	Cryptographic hash	SHA1	SHA, SHA256, MD5, SHA512, RIPEMD-160	
	Compression	Gzip	BZIP2, Lempel-Ziv-Oberhumer (LZO), MGzip	
	Dynamic sort	Heap sort	—	
LIBQUANTUM	Pauli Spin Op1	Sequential	Parallel	64
	Pauli Spin Op2	Sequential	Parallel	
	Not gate	Sequential	Parallel	
	C not gate	Sequential	Parallel	
	CC not gate	Sequential	Parallel	
	Arbitrary 1-bit Op	Sequential	Parallel	
VPR	FPGA placement	Simulated annealing	Deterministic parallel placement	2
BSDIFF	Suffix sort	Qsufsort	Divsufsort	6
	Compression	Bzip2	Gzip, LZO, MGzip	
FREQMINE	Frequent Itemset Mining (FIMI)	Fp-zhu	Afopt, Lcm	3
SCOTCH	Coarsen	Heavy-edge match	First-fit match	12
	Initial partition	FM	Gibbs-Poole-Stockmeyer method (GPS), greedy-graph-growing method (GGG)	
	Refinement	Band-FM	FM	

the versions, to highlight that there are multiple versions of important algorithms already available. Overall, our repository currently contains 51 different algorithm implementations, including multithreaded implementations, as well as SIMD or alternative sequential implementations. We organized the 51 algorithm implementations into 23 different SCGs. The number of implementations in each SCG varies from one (which means we didn't find compatible algorithms from the open-source domain) to six.

Componentization overhead

We first investigate the performance overhead of componentization. For this purpose, we compare the performance of the original (unmodified) sequential application against the componentized version. In this componentized version, no algorithm implementation has yet been substituted—the original program has simply been wrapped within components. As a result, the performance difference quantifies the overhead of componentizing a program. We define the overhead as the percentage increase in execution time of the componentized version over the original version. Figure 6 presents the measurements. The highest overhead is 1.9 percent for BZIP2, and all other programs exhibit only negligible overhead, if not slight performance improvements. For DEDUP,

Sources of Alternative Component Implementations

The following table lists the sources of all alternative implementations in Table 2. All of these implementations were taken from the public domain, which illustrates the power of SWAP for optimizing application performance by nonexpert programmers using existing component implementations written by experts.

Table A. Sources of the alternative implementations listed in Table 2.

Algorithms and implementations	Source
Parallel bidirectional Astar	Efficient obstacle avoidance using autonomously generated navigation meshes
Div-BWT	http://code.google.com/p/libdivsufsort
MTF-1 , MTF-2	http://nuwen.net/bwtzip.html
—	—
Arithmetic	http://michael.dipperstein.com/arithmetic/#download
SIMD-DCT	http://www.libjpeg-turbo.org/Main/HomePage
Fixed-size chunking	http://en.wikipedia.org/wiki/Data_deduplication
—	—
SHA, SHA256, MD5, SHA512, RIPEMD-160	http://www.openssl.org/source
BZIP2	http://www.bzip.org/downloads.html
LZO	http://www.oberhumer.com/opensource/lzo
MGzip	http://www.lemley.net/mgzip.html
—	—
Parallel Pauli Spin Op1	http://code.google.com/p/google-summer-of-code-2008-google/downloads/list
Parallel Pauli Spin Op2	
Parallel Not gate	
Parallel C not gate	
Parallel CC not gate	
Parallel Arbitrary 1-bit Op	
Deterministic parallel placement	Scalable and deterministic timing-driven parallel placement for FPGAs
Divsufsort	http://code.google.com/p/libdivsufsort
Gzip	http://www.gzip.org
LZO	http://www.oberhumer.com/opensource/lzo
MGzip	http://www.lemley.net/mgzip.html
Afopt, Lcm	http://fimi.ua.ac.be
First-fit match	http://www.labri.fr/perso/pelegrin/scotch
GPS, GGG	
FM	

the componentized version is actually faster because we modified the interface to pass the members of a structure directly to a function instead of the structure itself.

Substituting with parallel implementations

Using SWAP, we generated multithreaded versions of five programs (ASTAR, BZIP2, DEDUP, LIBQUANTUM, and VPR) starting from sequential implementations. The speedups are computed over the original (unmodified) program version, and they range from 1.57× to 2.79× (see Figure 7). All programs use four threads, but the parallel ASTAR algorithm is designed for two threads only, because it parallelizes a path search by starting from both ends of a path. We infrequently used quantitative specification. Only the security hash algorithm in DEDUP used it (see Figure 2). It means programming with SCG is at least no more complex than calling a parallel library, in terms of code complexity.

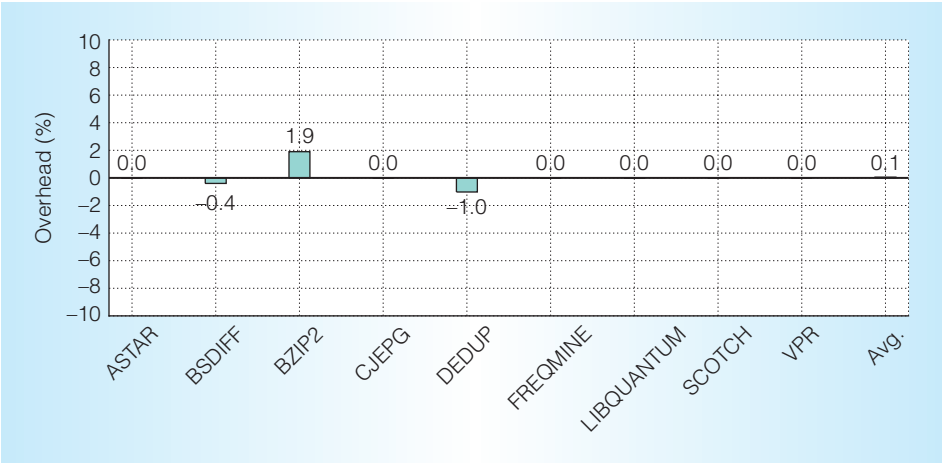

Figure 6. Overhead of componentization over original version. Other than BZIP2, all programs exhibit only negligible overhead, if not a slight performance improvement.

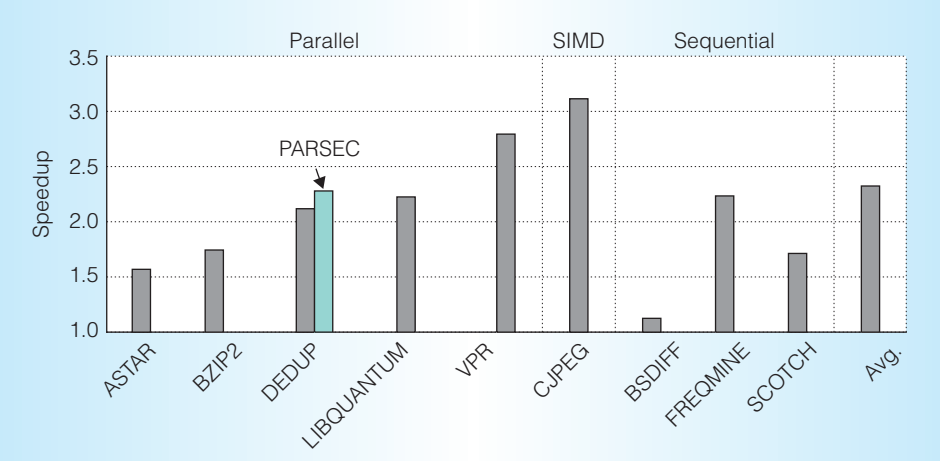

Figure 7. Performance improvement due to component substitution. We generated parallel versions for five benchmarks (on the left), a single-instruction, multiple-data (SIMD) version for one benchmark (in the middle), and three optimized sequential versions for three other benchmarks (on the right).

SWAP versus manual parallelization (DEDUP)

For DEDUP, instead of using the parallel implementation provided in the PARSEC benchmark suite, we started from the sequential version and substituted the original block compression with a parallel compression algorithm (MGzip). By varying the combinations of components, we created and evaluated 48 different implementations of DEDUP. The PARSEC parallel implementation resorts to pipeline parallelism, whereas we perform algorithm-based parallelization. In fact, in the PARSEC version, the compression stage is by far the most time-consuming, so there's one limiting stage in the whole execution pipeline. As a result, the benefit of application-based parallelization over algorithm-based parallelization is moderate (see Figure 7): the speedup is 2.12× (four threads) for algorithm-based parallelization versus 2.28× for application-based parallelization (see the PARSEC label pointing to the light gray bar for DEDUP). In general, application-based parallelization

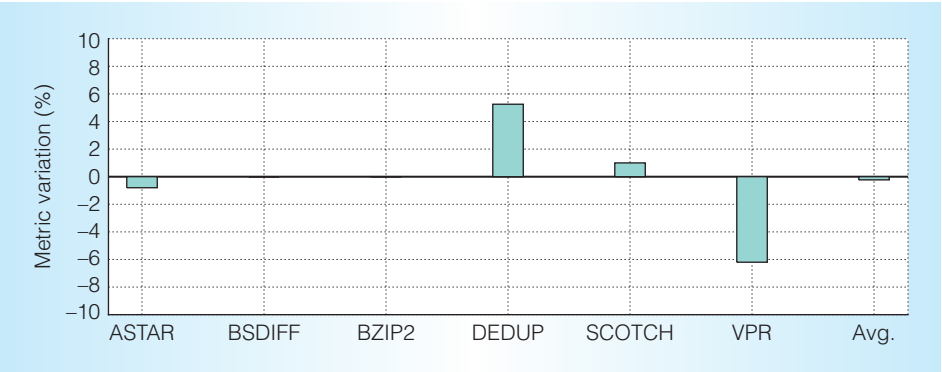

Figure 8. Impact of algorithm substitution on algorithm-specific metrics. Algorithm substitutions can improve or degrade algorithm-specific metrics, effectively illustrating a tradeoff between application speed and functional behavior.

should outperform algorithm-based parallelization, but many programmers aren't capable of conducting application-based parallelization, whereas algorithm-based parallelization can be transparent.

More than parallelization

Although our proposed framework's initial goal was to propose a parallelization approach for nonexpert programmers, it also provides more opportunities for performance improvement than just parallelization. We illustrate these additional applications in this article, but we leave their thorough investigation for further work.

Algorithm substitution

In some cases, just replacing a given algorithm with either an alternative implementation of the same algorithm or a compatible but different algorithm can yield significant performance improvements. We illustrate this point in Figure 7 (see "Sequential" substitutions), with three more benchmarks (BSDIFF, FREQMINE, and SCOTCH) listed in Table 1.

Hardware features

We can use the same concept to take advantage of special hardware features, such as SIMD, or accelerators, such as GPUs.[13] For instance, using SWAP, we substitute the sequential Discrete Cosine Transform (DCT) implementation of CJPEG with a SIMD-based DCT implementation, and we achieve a speedup of 3.11×, as shown in Figure 7.

Algorithm-specific metrics

As we mentioned earlier, programmers are also concerned with metrics other than performance. In some cases, they might want to bound these metrics so that algorithm substitution doesn't degrade them below a certain threshold, or they might accept to change them in order to further improve speed, or, on the contrary, they might want to optimize some of these metrics at the expense of speed. Among our benchmarks, ASTAR, BSDIFF, BZIP2, DEDUP, SCOTCH, and VPR exhibit such tradeoffs. In Figure 8, we report the percentage of variation of each application-specific metric (that is, average path length for ASTAR, patch size for BSDIFF, compression ratio for BZIP2 and DEDUP, graph cut size for SCOTCH, and bounding box size and critical-path delay for VPR) for the speedup results of Figure 7. The *y*-axis corresponds to the percentage of variation of the application-specific metric with respect to the original implementation. Positive metric variation means the framework can improve such metrics after swapping to get the performance improved. For instance, in DEDUP, although the speedup of algorithm-based parallelization (by SWAP) is a little lower than application-based parallelization (original PARSEC parallelization), the compression ratio is improved by 5.3 percent (see Figure 8). For VPR, we can get a speedup of 2.79× if we can tolerate 6.3 percent degradation of the critical-path delay by the place-and-route algorithm. Except for DEDUP and

VPR, the absolute variations are less than 1 percent.

Beyond the benefits for parallelization, we plan to extend the concept of substituting algorithms to supporting hardware accelerators in heterogeneous multicores: knowing which algorithm the program is performing lets us map the corresponding program section to a hardware accelerator (application-specific integrated circuit [ASIC], field-programmable gate array [FPGA], or GPU) if it can support that algorithm. MICRO

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. Lieven Eeckhout is supported through the FWO projects G.0232.06, G.0255.08, and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Community's Seventh Framework Program (FP7/2007-2013)/ERC Grant agreement no. 259295. Olivier Temam is supported by INRIA YOUHUA associated team funding. The remaining authors are supported by the National Natural Science Foundation of China under grants nos. 60921002 and 61033009, the National Basic Research Program of China under grant no. 2011CB302504, and the National Science and Technology Major Project of China under grant no. 2011ZX01028-001-002.

References

1. H. Kim et al., ''Scalable Speculative Parallelization on Commodity Clusters,'' *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2010, pp. 3-14.
2. M.W. Benabderrahmane et al., ''The Polyhedral Model Is More Widely Applicable Than You Think,'' *Compiler Construction,* Springer, 2010, pp. 283-303.
3. J. Reinders, *Intel Threading Building Blocks,* O'Reilly & Associates, 2007.
4. R. Blumofe et al., ''Cilk: An Efficient Multithreaded Runtime System,'' *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPOPP 95), ACM, 1995, pp. 207-216.
5. P. An et al., ''STAPL: An Adaptive, Generic Parallel C++ Library,'' *Proc. 14th Int'l Conf. Languages and Compilers for Parallel Computing,* Springer, 2001, pp. 193-208.
6. F. Putze, J. Singler, and P. Sanders, ''MCSTL: The Multi-core Standard Template Library,'' *Proc. Int'l European Conf. Parallel and Distributed Computing* (Euro-Par 07), Springer, 2007, pp. 682-694.
7. J. Ansel et al., ''PetaBricks: A Language and Compiler for Algorithmic Choice,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 09), ACM, 2009, pp. 38-49.
8. S. Sethumadhavan et al., ''COMPASS: A Community-Driven Parallelization Advisor for Sequential Software,'' *Proc. Int'l Workshop Multicore Software Eng.* (IWMSE 09), IEEE CS, 2009, pp. 41-48.
9. L. DeMichiel,, *Enterprise JavaBeans Specification, Version 2.1,* Sun Microsystems, Nov. 2003.
10. A. Rasche and A. Polze, ''Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET,'' *Proc. 6th IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing* (ISORC 03), IEEE CS, 2003, pp. 164-171.
11. J. Cheesman and J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software,* Addison-Wesley, 2000.
12. K. Lau and Z. Wang., ''Software Component Models,'' *IEEE Trans. Software Eng.,* Oct. 2007, pp. 709-724.
13. S.W. Keckler et al., ''GPUs and the Future of Parallel Computing,'' *IEEE Micro,* vol. 31, no. 5, 2011, pp. 7-17.

Hengjie Li is a PhD student in the State Key Laboratory of Computer Architecture of the Institute of Computing Technology, CAS (Chinese Academy of Sciences), and in the Graduate University of the Chinese Academy of Sciences. His research interests include performance optimization and parallelization. Li has an undergraduate degree in computer science and technology from Harbin Institute of Technology.

Wenting He is a PhD student in the State Key Laboratory of Computer Architecture of the Institute of Computing Technology,

CAS (Chinese Academy of Sciences), and in the Graduate University of the Chinese Academy of Sciences. Her research interests include performance optimization and approximate computation. He has an undergraduate degree in computer science from Fuzhou University.

Yang Chen is a postdoctoral researcher in the State Key Laboratory of Computer Architecture of the Institute of Computing Technology, CAS (Chinese Academy of Sciences). His research interests include performance optimization and iterative optimization. Chen has a PhD in computer science from the Institute of Computing Technology at the Chinese Academy of Sciences.

Lieven Eeckhout is an associate professor in the Electronics and Information Systems Department at Ghent University. His research interests include computer architecture and the hardware-software interface, with a focus on performance analysis, evaluation and modeling, and workload characterization. Eeckhout has a PhD in computer science and engineering from Ghent University. He is a member of IEEE and the ACM and the Associate Editor in Chief of *IEEE Micro*.

Olivier Temam is a senior research fellow at INRIA Saclay in Paris, where he heads the ByMoore Group, and an adjunct professor at École Polytechnique. His research interests include processor architecture and simulation, program optimization, alternative technologies, and computing paradigms. Temam has a PhD in computer science from the University of Rennes.

Chengyong Wu is a professor in the Institute of Computing Technology, CAS (Chinese Academy of Sciences). His research interests include parallel-programming environments and iterative optimization. Wu has a PhD in computer science from the Chinese Academy of Sciences. He's a member of the ACM and a senior member of the China Computer Federation (CCF).

Direct questions and comments about this article to Hengjie Li, No. 6 Kexueyuan South Road, Zhongguancun, Haidian District, Beijing, China, 100190; lihengjie@ict.ac.cn.